下载APP

24 | 容量场景之一：索引优化和Kubernetes资源分配不均衡怎么办？

2021-05-14 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 22:46 大小 20.86M

你好，我是高楼。

我们知道，做容量场景的目的是要回答“线上容量最大能达到多少”的问题，这就要求我们在设计和执行容量场景的时候要非常严谨。当然，这也意味着容量场景将是一个艰辛的过程。通过这节课，你就能深切地体会到。

今天，我们重点来解决索引优化和 Kubernetes 调度不均衡的问题。关于索引优化，你可能会奇怪，基准场景都浮过一遍了，为啥还有要看索引的问题？是呀，确实让人疑惑。这里就可以看出，容量场景和基准场景真的不太一样，因为这其中有业务相互影响的问题。

而 Kubernetes 调度不均衡的问题将导致多个 Pod 运行在了同一个 worker 上，像这样的问题，我们不在容量场景中是看不到的，希望能对你处理类似问题有一个借鉴。

此外，我们还将一起看看在压力稳定的情况下，响应时间不断攀升该怎么办。这种问题很常见，但是每次出现问题点都不太相同，这次你将看到一个具体的案例。

好，我们开始吧！

场景运行数据

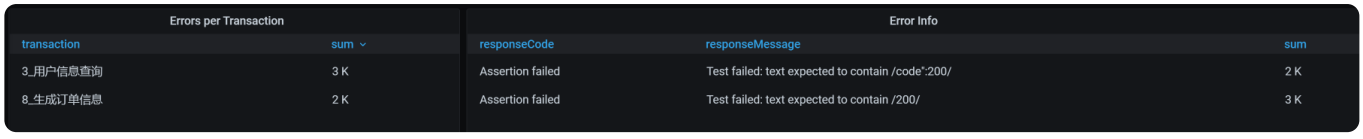
第一次运行

不得不承认，第一次来到容量场景，还真是心惊胆颤的。

首先，我们小心翼翼地设置起容量场景的比例，也就是我们在第 5 讲中提到的业务比例，再设置好相应的参数和关联。然后，我们把容量场景跑起来，得到了这样的信息：

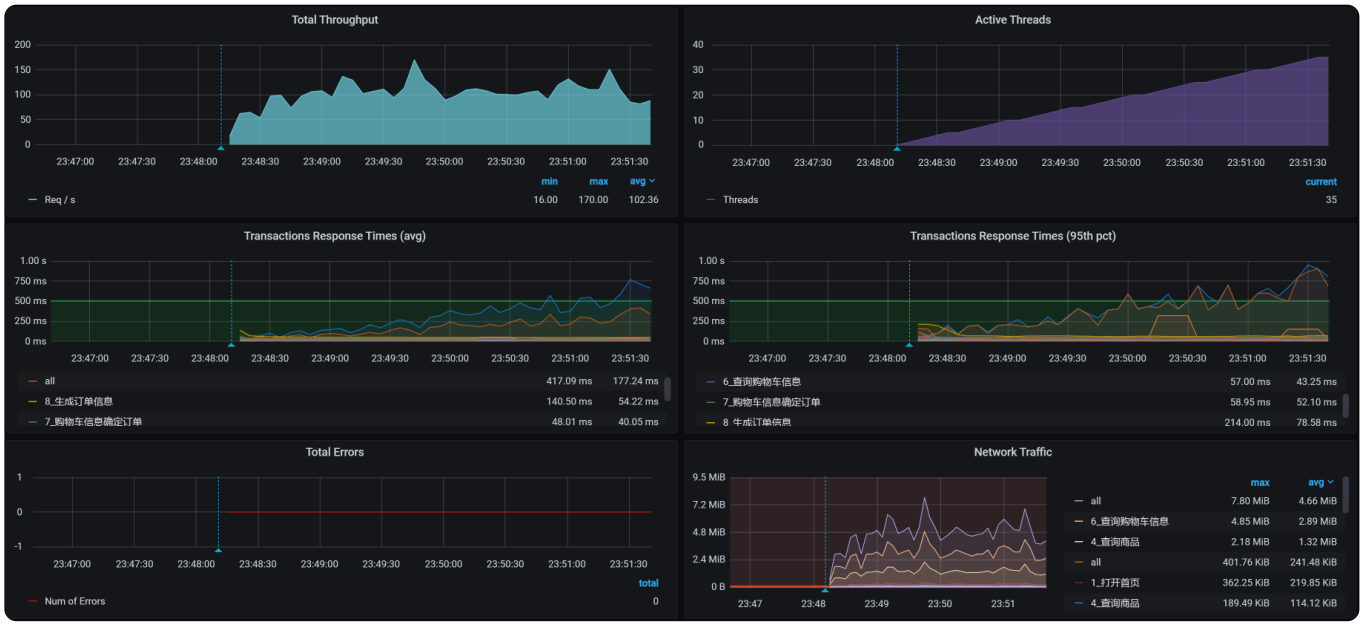
顿时就有一种满头包的感觉，有没有？！不过，“见招拆招，遇魔降魔”不就是我们的宗旨吗？既然有错，那咱们就先解决错误吧。

我们先看下错误信息：



看来是两个脚本有问题。我单独运行了这两个脚本之后，发现是参数化数据设置错了。因为之前每个接口都是单脚本运行的，而现在要连到一起跑，所以，参数化数据需要重新配置。这种简单的错误，我就不详细描述了，只要你细心一点，很快就能查到。

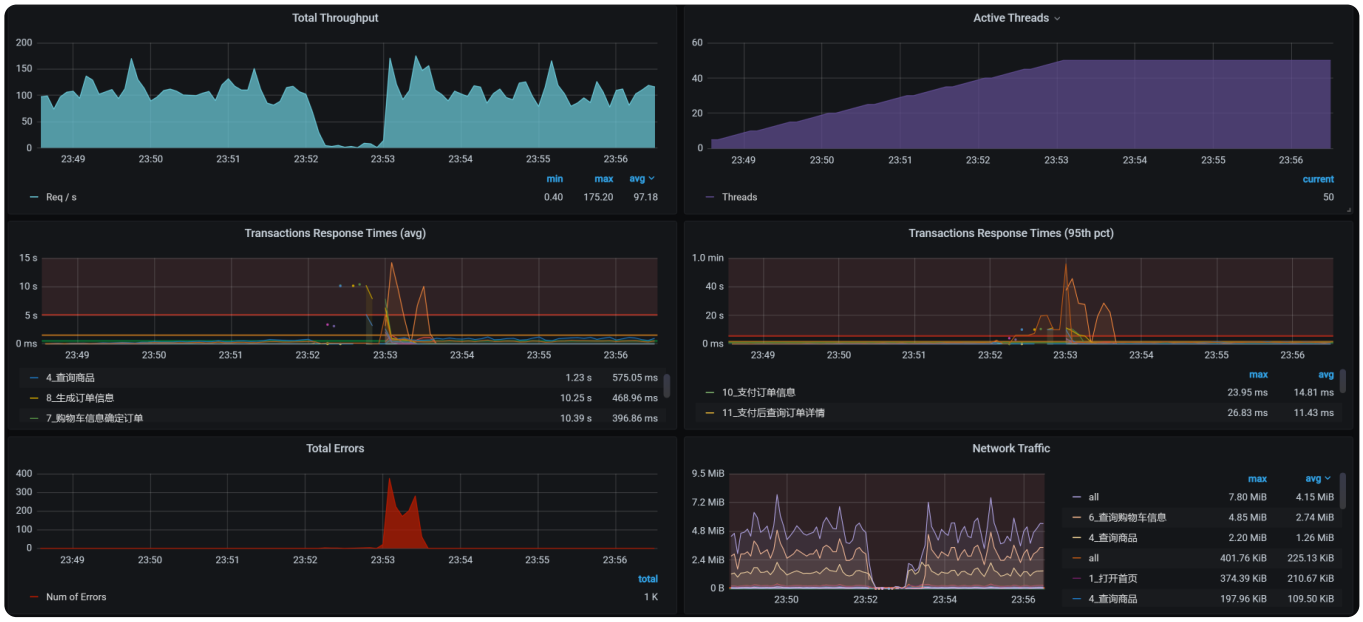
修改了这个简单的脚本错误之后，我们再把容量场景跑起来：



先不管性能怎么样，你看，现在至少没有错误信息了，对不对？很好，我们接着来看容量场景中的其他问题。

第二次运行

解决了第一个问题之后，我们把场景运行一段时间，又看到了下面这样的场景运行数据：



数据并没有太好看，还是有失败的地方。我们来看看是怎么一回事：

Errors per Transaction

transaction	sum
5_商品加入购物车	235
3_用户信息查询	194
2_用户登陆	151
11_支付后查询订单详情	151
8_生成订单信息	143
6_查询购物车信息	141
9_支付前查询订单列表	124
7_购物车信息确定订单	122
10_支付订单信息	91

Error Info

responseCode	responseMessage	sum
Assertion failed	Test failed: text expected to contain /token/	151
Assertion failed	Test failed: text expected to contain /pageNum/	124
Assertion failed	Test failed: text expected to contain /code":200/	761
Assertion failed	Test failed: text expected to contain /cartPromotionItemList/	122
Assertion failed	Test failed: text expected to contain /200/	194

从数据上来看，错误信息应该和脚本无关，是某个组件出现了问题，导致所有的脚本都不能正常运行了。

通过查看全局监控的 Pod 界面，我们看到这样的现象：

Pod名称	CPU%(总核)	使用核数	CPU需求	CPU限制	WSS%	WSS	RSS%	RSS	内存需求	内存限制	磁盘	命名空间	重启
elasticsearch-data-0	99.42%	0.99	1.00	1.00	85.09%	1.70 GiB	70.18%	1.40 GiB	2.00 GiB	2.00 GiB	619.54 MiB	default	0

考虑到我们在前面分析中对 ES 的资源限制得比较狠（只有 1C，因为不想让 ES 影响其他业务，这一点在第 15 讲中已经有过分析），在前文中虽然我们通过对 ES POD 扩容提升了性能，但为了不影响后面的优化，我又给调回去去了。在这里，我们先把使用 ES 的业务，也就是查询商品给禁掉，再次运行起来容量场景。

第三次运行

我们来看第三次运行的结果：

还是有少量的报错，我们先看错在哪里，再解决响应时间长的问题。

这里我说明一点，对于报错，我们可以先看错误信息是什么，脚本有没有问题。但是，由于我们已经来到了容量场景，脚本如果有问题，就会在基准场景中体现出来了。所以，这时候我们去查应用的日志更为理智。

于是，通过一层层地查日志，我们留意到了 Cart 信息：

```
1 2021-01-26 00:20:43.585 ERROR 1 --- [o-8086-exec-669] o.a.c.c.C.[.[.[/].[dispa
2 java.net.SocketException: Socket closed
```

复制代码

很显然，在 Cart 服务上，我们看到已经报远程调用（Feign 调用）超时了，并且调用的是 member 服务。

由于是 Cart 服务调用 Member 服务出的错，我们现在去看 Member 日志：

```
1 2021-01-26 00:20:46.094 ERROR 1 --- [o-8083-exec-308] o.a.c.c.C.[.[.[/].[dispa
2 java.net.SocketTimeoutException: timeout
```

复制代码

你看，Member 上显示的是读 Auth 出错。可是，当我再去看 Auth 服务时，发现 Auth 服务上空空如也，啥错也没有。

根据我的经验，报错可能是由于一个瞬间问题导致的读超时，只有等它再次出现我们再去收拾了。

因为对于错误信息，我们暂时还没查到具体是什么问题。所以，我们接着持续运行场景，我们看一下结果：

你瞧，确实有少量报错。从整体上看，这只是少量的读超时问题。在性能测试中，有毛刺是很正常的情况。而且现在的部署结构中仍然有单点 Java 应用存在，这种情况也不可避免。

不过，从上图中，我们能看到一个比较明显的问题，那就是 TPS 在不断下降，我们得先解决这个问题才行。

第一阶段分析

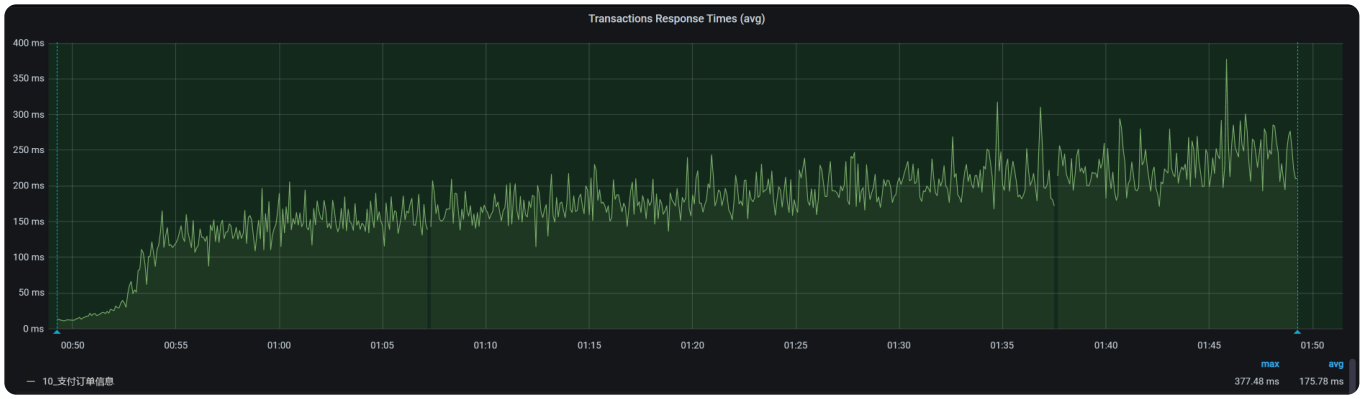
定向监控分析

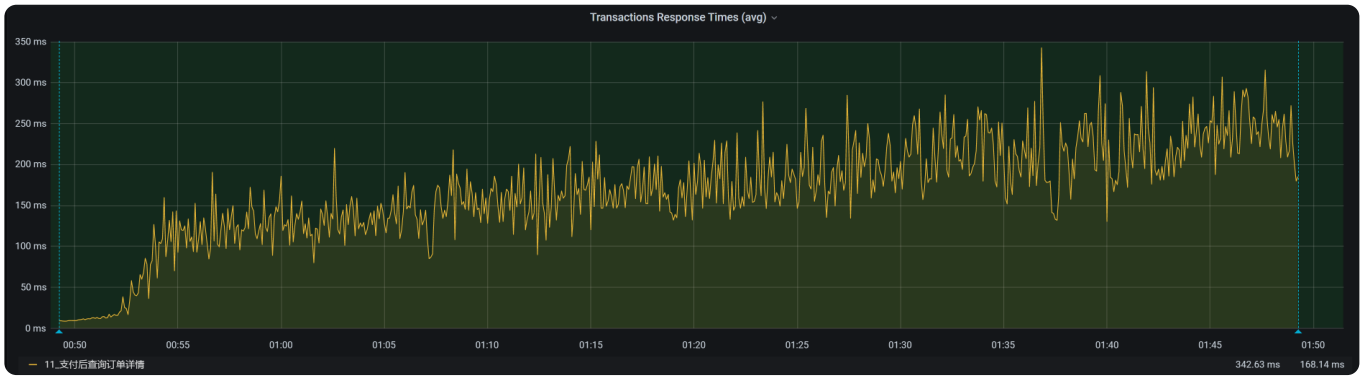
虽然整体的 TPS 不高，但是我们面对的第一个问题仍然是：TPS 会在压力持续之下不断下降。

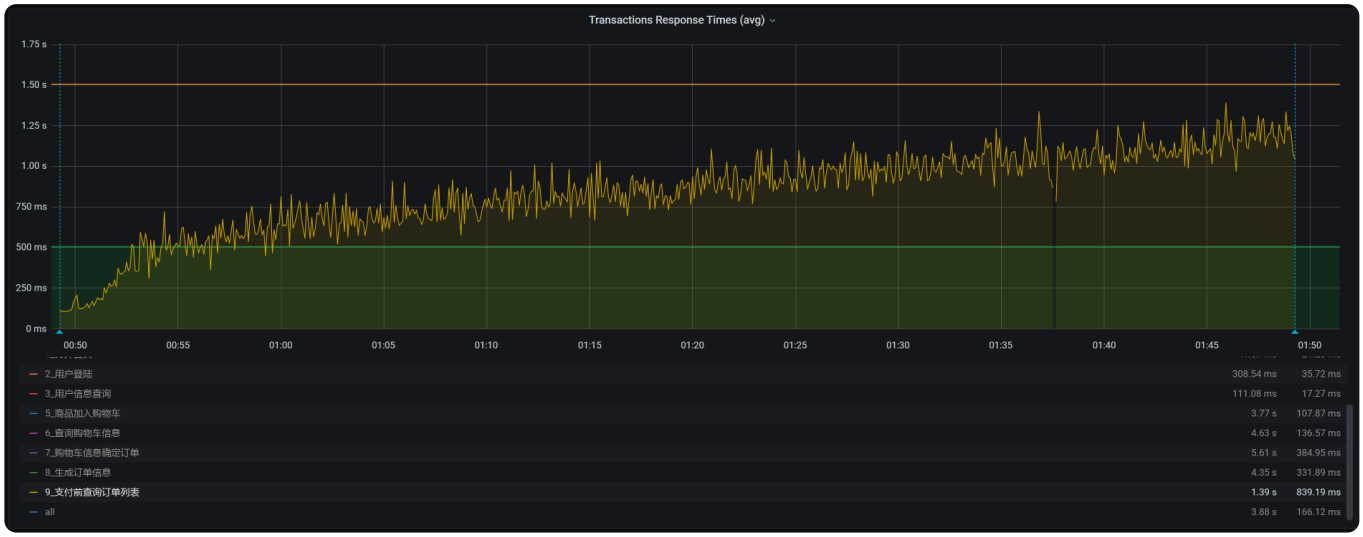
由于我们现在执行的是容量场景，包含了多个业务，所以我们得先知道是什么样的业务导致了像上图中那样的 TPS 趋势。而容量场景中的脚本都是接口串行的，因此，当一个脚本出现这种情况时，所有的业务都会慢下来，即使是那些本身不慢的业务，TPS 也会受到影响。

但是，对于没有问题的业务接口，它们的响应时间是不会受到影响的。那些响应时间变长的业务接口，自然就成了我们重点关注的对象了。

通过一个一个的业务接口排查，我发现有三个业务的响应时间，随着场景的持续在不断增加：







这三个业务分别是“支付前查询订单列表”、“支付订单信息”和“支付后查询订单详情”，但是在前面的基准场景的分析中，它们并没有出现这种现象，也就是说它们是在容量场景中新出现了问题。

既然这几个业务的脚本都有这种现象，那我们就接着来分析时间具体慢在了哪里。我先从最简单的业务链路 paySuccess（支付订单信息）接口查起。

这个接口的架构图我们在第 22 讲中已经讲过了，就是 JMeter - Gateway - Order - Mysql（如果再加上其他的技术组件，架构图会更复杂一些。不过，我们现在还没有看到它们的影响，所以，我暂时不列在这里）。

通过性能分析决策树的全局监控计数器逐层查看，我看到有一个 SQL 语句在不断变慢：

复制代码

```
1 SELECT id, member_id, coupon_id, order_sn, create_time, member_username, total
```

接着，我们进一步看看这个 Select 语句的索引：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	oms_order	(Null)	ref	member_id,member_id_2	member	8	const	67530	10	Using index condition; Using where; Using filesort

咦，怎么有两个可能会用到的索引？具体查看一下，果然有两个索引，并且还是在同一个列上。

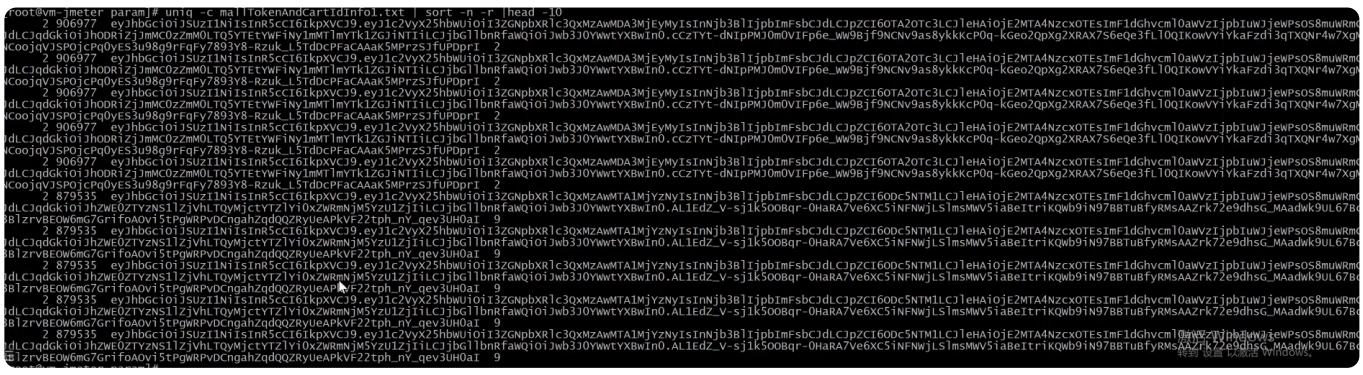
Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_c
oms_orde	0	PRIMARY	1	id	A	12080675	(Null)	(Null)		BTREE		
oms_orde	1	member_id	1	member_id	A	2467043	(Null)	(Null)		BTREE		
oms_orde	1	member_id_2	1	member_id	A	2467043	(Null)	(Null)		BTREE		

我们先删一个再说。不过，删索引并不是为了解决性能问题，我们只是顺手改一下而已。

现在，索引上显示的数据很明确，随着容量场景的执行，这个 SQL 查的数据行数越来越多。既然这个 Select 语句越来越慢，那我们就去查一下根据它的条件产生的直方图是什么。我查到了这样的直方图：

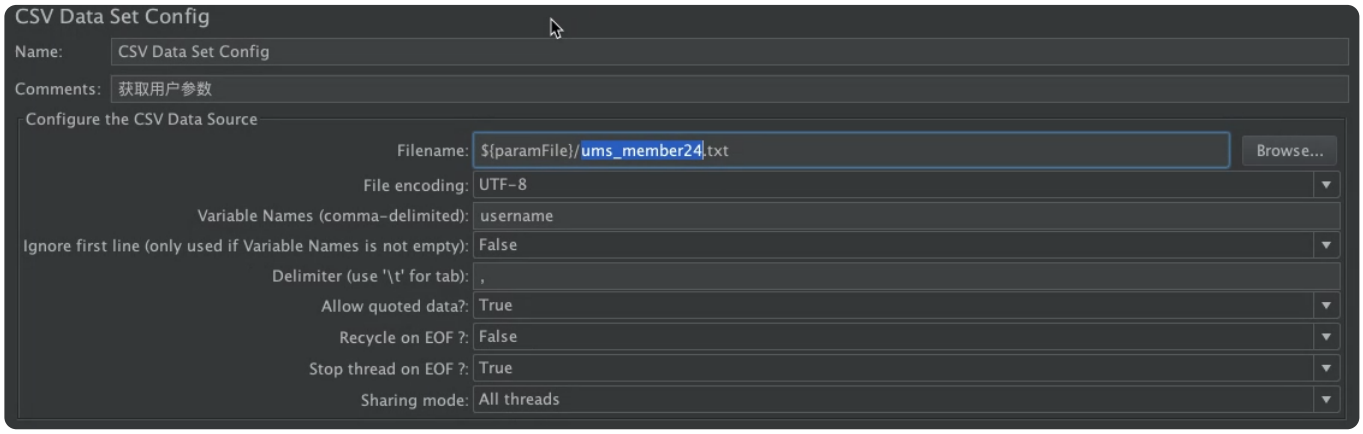
member_id	delete_status	count(*)
906977	0	646605
856977	0	646073
683597	0	645894
547363	0	645683
648423	0	645683
541675	0	645463
559328	0	645396
1133681	0	645394
593529	0	645386
846849	0	645350
819860	0	645268
914466	0	645254
519056	0	644887
604261	0	644739
569886	0	644727
597427	0	644624
879535	0	644586
443436	0	644409
812008	0	644087
1110922	0	644028
287019	0	34458
288682	0	34454
383268	0	34453
389004	0	34449
421740	0	34441

你看，一个用户下出现了很多个订单，应该是出现了集中插入订单数据的情况。因此，我们查一下在使用参数化的文件中，用户信息是不是也重复了。

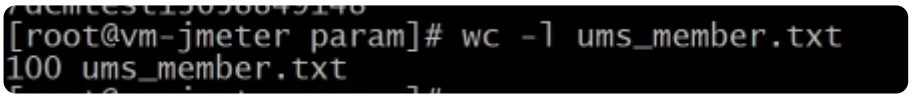


通过查找，我们看到在参数化文本中，确实有大量的重复数据。我们梳理一下业务脚本的参数化逻辑：

既然订单数据是重复的，那我们就反向追溯回去，肯定是购物车数据重复了；再接着往前追溯，应该是 member_id 在使用用户 Token 时出现了重复。在检查了所有的业务脚本之后，我们看到是获取用户参数脚本生成的 Token：



而这个脚本中的数据，使用的是这样的参数文件：



这个参数文件中只有 100 条数据，而这 100 条数据中就有大量的重复 member_ID。这样一来，问题就清楚了：原来是在造数据阶段，我们一开始使用的 Member 数据已经有了问题，从而导致了订单数据大量重复。

问题的原因查清楚了，我们现在只有重新造数据了。这时候我们就要注意，必须使用不重复的 member_id 来造订单数据，并且在每一步中，检查一下生成的数据是否有重复的现象。

经过一番折腾之后，TPS 如下：

这个优化效果很好，我们通过解决参数化的重复问题，解决了同一个用户下产生大量订单的问题，进而解决了我们在前面看到的响应时间不断增加的问题。

对于这样的参数化问题，其实难点就在于参数化是登录时做的，而在订单脚本中使用的参数，会用到登录脚本中产生的数据，但由于前面做的参数化出现了重复，导致后续的参数也重复了。对于这样的问题，我们需要耐心梳理参数化的数据来源。

到这里，我们解决了响应时间不断增加的问题，不过，我们的优化还没结束。你看上图中响应时间的窗口，出现了有的接口响应时间较长、整体 TPS 并没有太高的现象，因此，我们还要进行第二阶段的分析。

第二阶段分析

根据我们的性能分析逻辑，下面我们要做的就是拆分响应时间和查看全局监控数据了，并且这两步都要做。

相信你已经发现了，我们在分析的时候，有时候要拆分响应时间，有时候就直接看全局监控数据了。这是因为**当我们能从全局监控数据中看到问题所在时，不用拆分响应时间就能往下分析**，我们只需要按照分析七步法，接着往下走就可以了。

如果我们没有从全局监控数据中看到比较明显的资源消耗，同时，响应时间又在不断上升，那我们就必须拆分响应时间，来精准地判断。

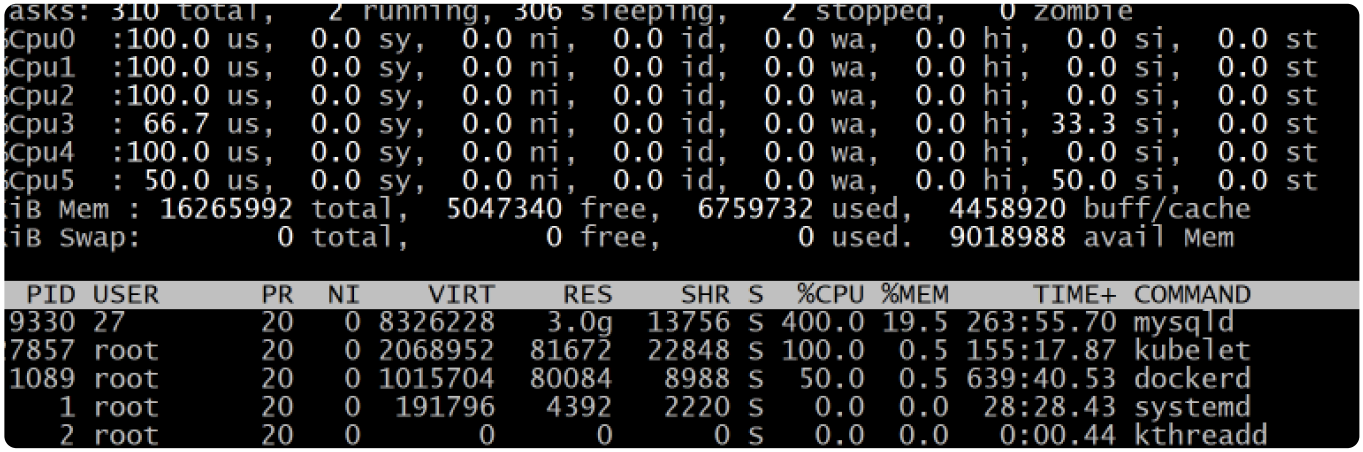
全局监控分析

在查看了全局监控数据之后，数据库的资源如下所示：

服务器资源总览表【JOB：node-exporter，主机总数：12】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.46 week	7.64 GiB	4	0.49	12.37%	35.89%	29.95%	9.21 kB/s	88.77 kB/s	312	22	394.41 kbps	394.41 kbps
k8s-master-2	k8s-master-2	8.16 week	7.64 GiB	4	0.65	20.16%	23.44%	29.95%	0 B/s	84.59 kB/s	259	21	284.72 kbps	413.49 kbps
k8s-master-3	k8s-master-3	1.46 week	7.64 GiB	4	0.33	11.47%	34.23%	29.95%	9.21 kB/s	87.11 kB/s	303	91	244.95 kbps	470.46 kbps
k8s-worker-1	k8s-worker-1	1.46 week	15.51 GiB	6	25.40	98.07%	44.58%	29.95%	32.22 kB/s	1.78 MB/s	28	139	22.96 Mbps	23.21 Mbps
k8s-worker-2	k8s-worker-2	6.98 hour	15.51 GiB	6	1.85	27.62%	18.95%	33.89%	13.81 kB/s	56.69 kB/s	28	328	9.10 Mbps	8.85 Mbps
k8s-worker-3	k8s-worker-3	1.46 week	15.51 GiB	6	2.76	40.63%	28.21%	45.86%	23.02 kB/s	97.41 kB/s	29	326	21.02 Mbps	19.55 Mbps
k8s-worker-4	k8s-worker-4	1.15 week	15.51 GiB	6	3.10	38.68%	30.39%	36.27%	9.21 kB/s	166.44 kB/s	48	21 K	15.04 Mbps	15.22 Mbps
k8s-worker-5	k8s-worker-5	7.07 week	15.51 GiB	6	2.83	42.70%	55.53%	37.88%	32.22 kB/s	9.80 kB/s	28	145	17.15 Mbps	19.27 Mbps
k8s-worker-6	k8s-worker-6	7.07 week	15.51 GiB	6	1.08	13.28%	22.71%	40.23%	13.81 kB/s	3.26 kB/s	28	129	752.35 kbps	723.31 kbps
k8s-worker-7	k8s-worker-7	6.77 week	15.51 GiB	4	1.20	25.44%	30.38%	42.30%	13.81 kB/s	26.71 kB/s	30	214	7.53 Mbps	7.40 Mbps
k8s-worker-8	k8s-worker-8	5.59 week	15.51 GiB	4	1.15	11.92%	32.96%	56.23%	0 B/s	3.72 kB/s	27	199	21.41 Mbps	21.50 Mbps
k8s-worker-9	k8s-worker-9	6.77 week	15.51 GiB	6	2.12	24.24%	27.48%	29.95%	4.60 kB/s	6.17 kB/s	28	85	31.49 Mbps	45.81 Mbps

你看，worker-1 的 CPU 资源使用率比较高，我们进到这个机器中看一下 top：



```
asks: 310 total,   2 running, 306 sleeping,   2 stopped,   0 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 66.7 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi, 33.3 si,  0.0 st
%Cpu4  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  : 50.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi, 50.0 si,  0.0 st
KiB Mem : 16265992 total,  5047340 free,  6759732 used,  4458920 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  9018988 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 9330 27        20   0 8326228   3.0g  13756 S 400.0 19.5 263:55.70 mysqld
 7857 root      20   0 2068952  81672  22848 S 100.0  0.5 155:17.87 kubelet
 1089 root      20   0 1015704  80084   8988 S  50.0  0.5 639:40.53 dockerd
    1 root      20   0  191796   4392   2220 S   0.0  0.0  28:28.43 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.44 kthreadd
```

显然多个 CPU 使用率已经达到了 100%，从 process table 中我们也能看到，CPU 消耗最高的显然是 MySQL 进程。那下面的的分析逻辑就很清楚了（在第 16 讲中已经详细描述过）。

定向监控分析

这是一个典型的数据库 CPU 使用率高的问题，而且也相对比较简单。我们仍然是看 MySQL 的全局监控数据、查 SQL、查执行计划和创建索引。有问题的 SQL 如下所示：

```
EXPLAIN SELECT
  p.id,
  p.`name`,
  p.promotion_type,
  p.gift_growth,
  p.gift_point,
  sku.id sku_id,
  sku.price sku_price,
  sku.sku_code sku_sku_code,
  sku.promotion_price sku_promotion_price,
  sku.stock sku_stock,
  sku.lock_stock sku_lock_stock,
  ladder.id ladder_id,
  ladder.count ladder_count,
  ladder.discount ladder_discount,
  full_re.id full_id,
  full_re.full_price full_full_price,
  full_re.reduce_price full_reduce_price
FROM
  pms_product p
  LEFT JOIN pms_sku_stock sku
    ON p.id = sku.product_id
  LEFT JOIN pms_product_ladder ladder
    ON p.id = ladder.product_id
  LEFT JOIN pms_product_full_reduction full_re
    ON p.id = full_re.product_id
WHERE p.id IN (338)
```

1 结果 | 2 信息 | 3 表数据 | 4 信息

(只读)

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	p	(NULL)	const	PRIMARY	PRIMARY	8	const	1	100.00	(NULL)
1	SIMPLE	sku	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	113467	100.00	Using where
1	SIMPLE	ladder	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	26	100.00	Using where; Using join buffer (Block Nested Loop)
1	SIMPLE	full_re	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	26	100.00	Using where; Using join buffer (Block Nested Loop)

同样，type 列的 ALL 告诉我们这个 SQL 是全表扫描。在相应的表上创建索引之后，执行计划如下：

```
EXPLAIN
SELECT
  p.id,
  p.`name`,
  p.promotion_type,
  p.gift_growth,
  p.gift_point,
  sku.id sku_id,
  sku.price sku_price,
  sku.sku_code sku_sku_code,
  sku.promotion_price sku_promotion_price,
  sku.stock sku_stock,
  sku.lock_stock sku_lock_stock,
  ladder.id ladder_id,
  ladder.count ladder_count,
  ladder.discount ladder_discount,
  full_re.id full_id,
  full_re.full_price full_full_price,
  full_re.reduce_price full_reduce_price
FROM
  pms_product p
  LEFT JOIN pms_sku_stock sku
    ON p.id = sku.product_id
  LEFT JOIN pms_product_ladder ladder
    ON p.id = ladder.product_id
  LEFT JOIN pms_product_full_reduction full_re
    ON p.id = full_re.product_id
WHERE p.id IN (250);

SHOW INDEX FROM pms_sku_stock;
ALTER TABLE `pms_sku_stock` ADD INDEX index_name ( `product_id` ) ;

SHOW INDEX FROM pms_product_ladder;
ALTER TABLE `pms_product_ladder` ADD INDEX index_name ( `product_id` ) ;

SHOW INDEX FROM  pms_product_full_reduction;
ALTER TABLE `pms_product_full_reduction` ADD INDEX index_name ( `product_id` ) ;
```

1 结果 | 2 信息 | 3 表数据 | 4 信息

(只读)

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	p	(NULL)	const	PRIMARY	PRIMARY	8	const	1	100.00	(NULL)
1	SIMPLE	sku	(NULL)	ref	index_name	index_name	9	const	1	100.00	(NULL)
1	SIMPLE	ladder	(NULL)	ref	index_name	index_name	9	const	1	100.00	(NULL)
1	SIMPLE	full_re	(NULL)	ref	index_name	index_name	9	const	1	100.00	(NULL)

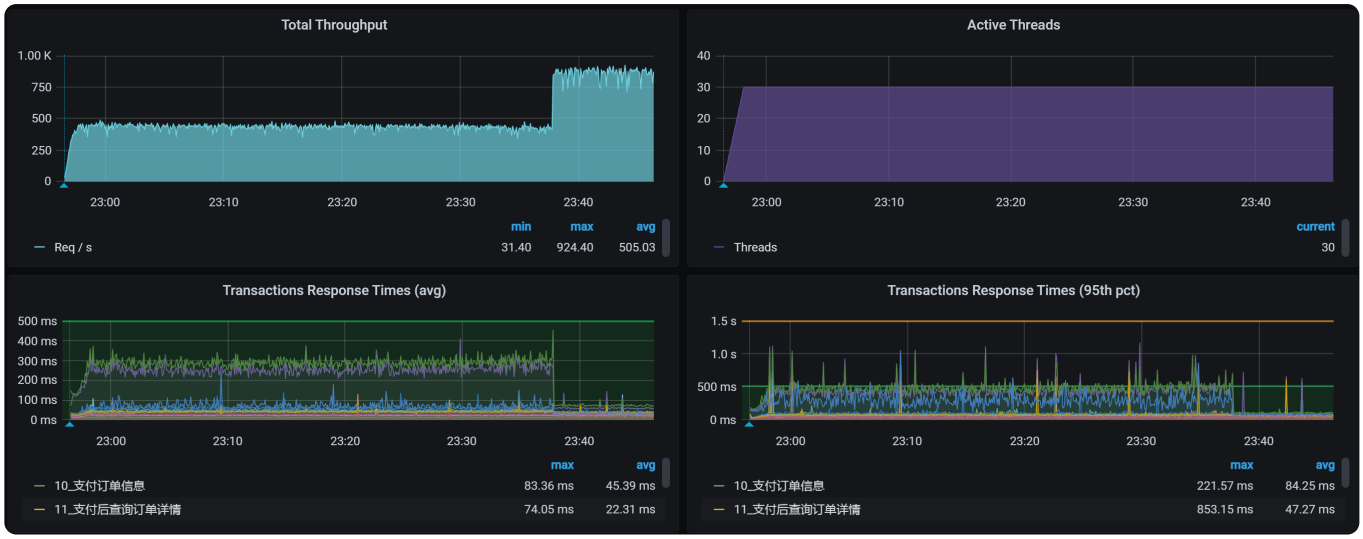
从 type 中的 ref 值来看，我们现在已经创建好索引了。

接着我们把容量场景再执行起来，我们得到下面这张全局监控图：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.46 week	7.64 GiB	4	0.36	11.68%	35.87%	29.95%	13.81 kB/s	92.63 kB/s	313	23	394.09 kbps	394.09 kbps
k8s-master-2	k8s-master-2	8.16 week	7.64 GiB	4	0.61	19.86%	23.42%	29.95%	0 B/s	89.47 kB/s	259	25	274.07 kbps	410.70 kbps
k8s-master-3	k8s-master-3	1.46 week	7.64 GiB	4	1.42	11.59%	34.17%	29.95%	4.60 kB/s	92.85 kB/s	302	96	260.54 kbps	464.22 kbps
k8s-worker-1	k8s-worker-1	1.46 week	15.51 GiB	6	8.38	44.60%	44.87%	29.95%	0 B/s	3.45 MB/s	31	164	44.71 Mbps	45.05 Mbps
k8s-worker-2	k8s-worker-2	7.31 hour	15.51 GiB	6	3.09	41.69%	19.30%	34.17%	4.76 kB/s	96.69 kB/s	29	492	17.59 Mbps	17.19 Mbps
k8s-worker-3	k8s-worker-3	1.46 week	15.51 GiB	6	5.61	61.51%	28.23%	46.18%	32.22 kB/s	150.21 kB/s	29	544	40.66 Mbps	31.48 Mbps
k8s-worker-4	k8s-worker-4	1.16 week	15.51 GiB	6	7.00	58.05%	30.50%	36.40%	39.01 B/s	328.45 kB/s	48	26 K	27.63 Mbps	26.84 Mbps
k8s-worker-5	k8s-worker-5	7.07 week	15.51 GiB	6	4.35	38.66%	55.93%	37.89%	25.24 kB/s	9.12 kB/s	28	156	15.62 Mbps	19.39 Mbps
k8s-worker-6	k8s-worker-6	7.07 week	15.51 GiB	6	0.69	13.13%	22.76%	40.23%	27.62 kB/s	4.07 kB/s	28	115	748.13 kbps	720.95 kbps
k8s-worker-7	k8s-worker-7	6.77 week	15.51 GiB	4	2.36	38.97%	30.36%	42.31%	4.60 kB/s	46.17 kB/s	30	285	14.51 Mbps	14.39 Mbps
k8s-worker-8	k8s-worker-8	5.59 week	15.51 GiB	4	0.61	11.96%	32.95%	56.23%	0 B/s	3.72 kB/s	27	188	16.50 Mbps	16.59 Mbps
k8s-worker-9	k8s-worker-9	6.77 week	15.51 GiB	6	2.38	28.29%	27.51%	29.95%	9.21 kB/s	7.95 kB/s	28	75	67.63 Mbps	36.19 Mbps

可以看到，worker-1 的 CPU 已经下去了。

我们再来看一下场景的运行结果：



嗯，效果还是不错的。索引能带来的作用就是这样，TPS 会在我们增加索引之后，增加很多。这一点我在之前的课程中已经反复提到了，这里就不再详细说了。

有人可能会问，不对呀，在之前的基准场景中为什么没有增加索引呢？之前的接口难道没有因为没增加索引出现瓶颈吗？

对此，我其实也有这样的疑问。你想想，这个 SQL 是在生成订单时使用的，和优惠卷相关，我删除了索引之后，又回去重新执行了生成订单的接口，发现 TPS 还是能达到之前基准场景的峰值。可见，这个 SQL 的问题在之前的场景中，确实没有表现出来，在混合场景中才会出现。

在加了索引之后，我以为 TPS 可以正常一会了。可是没想到，第二天我在执行场景时，还是看到了不想看到的结果，具体问题我们在下一个阶段分析。

这个项目就是这样，步步为坑。不过，也正是这种步步为坑的项目，才让我们有了更多的分析机会。下面我们就来看看到底出了什么问题。

第三阶段分析

全局监控分析

这就是在我持续执行容量场景之后，全局监控的结果：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.59 week	7.64 GiB	4	3.70	36.80%	35.91%	29.95%	4.60 kB/s	89.64 kB/s	307	30	375.65 kbps	375.65 kbps
k8s-master-2	k8s-master-2	8.29 week	7.64 GiB	4	0.78	18.64%	23.24%	29.95%	0 B/s	86.19 kB/s	261	23	287.23 kbps	408.11 kbps
k8s-master-3	k8s-master-3	1.59 week	7.64 GiB	4	1.70	36.86%	35.27%	29.95%	27.62 kB/s	88.72 kB/s	301	87	233.45 kbps	467.55 kbps
k8s-worker-1	k8s-worker-1	1.59 week	15.51 GiB	6	11.15	50.74%	57.34%	35.36%	4.60 kB/s	2.35 MB/s	33	140	25.86 Mbps	26.00 Mbps
k8s-worker-2	k8s-worker-2	1.21 day	15.51 GiB	6	9.34	55.64%	24.93%	36.82%	9.21 kB/s	62.62 kB/s	27	413	9.91 Mbps	5.20 Mbps
k8s-worker-3	k8s-worker-3	1.59 week	15.51 GiB	6	39.14	93.89%	45.05%	51.30%	23.02 kB/s	114.62 kB/s	30	753	24.66 Mbps	18.79 Mbps
k8s-worker-4	k8s-worker-4	1.29 week	15.51 GiB	6	9.54	73.36%	31.92%	39.21%	23.02 kB/s	190.95 kB/s	49	20 K	16.77 Mbps	16.72 Mbps
k8s-worker-5	k8s-worker-5	7.20 week	15.51 GiB	6	2.76	22.35%	66.02%	37.98%	18.41 kB/s	8.11 kB/s	28	143	5.75 Mbps	7.02 Mbps
k8s-worker-6	k8s-worker-6	7.20 week	15.51 GiB	6	0.46	12.54%	22.71%	40.25%	13.81 kB/s	4.10 kB/s	29	115	916.54 kbps	884.33 kbps
k8s-worker-7	k8s-worker-7	6.90 week	15.51 GiB	4	1.86	30.71%	34.99%	46.20%	13.81 kB/s	25.74 kB/s	28	305	8.76 Mbps	6.25 Mbps
k8s-worker-8	k8s-worker-8	5.72 week	15.51 GiB	4	0.24	10.20%	33.00%	56.33%	0 B/s	5.02 kB/s	27	189	4.79 Mbps	4.71 Mbps
k8s-worker-9	k8s-worker-9	6.90 week	15.51 GiB	6	3.31	38.56%	27.89%	29.95%	14.51 kB/s	7.73 kB/s	29	82	626.36 Mbps	22.11 Mbps

注意，这里不是说不需要拆分响应时间，而是我已经看过了响应时间的拆分数据，发现是 Order 服务上消耗的时间多。鉴于我们在前面的课程里已经做过多次响应时间的拆分，所以，我在这里就不再具体列出数据了。

看到全局监控数据的第一个界面，我们就已经可以发现资源消耗比较高的节点了。从数据来看，worker-3 的 CPU 使用率上来了达到了 90% 以上，worker-4 的 CPU 使用率也上来了达到了 70% 以上，也就是说 TPS 会不断地掉下来。下面我带着你来分析一下。

定向监控分析

我们进入到 worker-3 中，执行 top，看看它上面都跑了哪些服务：

复制代码

```
top - 23:21:36 up 11 days,  5:37,  5 users,  load average: 40.53, 49.79, 53.30
Tasks: 335 total,   1 running, 332 sleeping,   2 stopped,   0 zombie
%Cpu(s): 82.5 us,  8.7 sy,  0.0 ni, 3.0 id,  0.0 wa,  0.0 hi,  3.2 si,  2.6 st
KiB Mem : 16265984 total,  2802952 free,  7089284 used,  6373748 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  8759980 avail Mem

```

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26573 root      20   0 8930960 837008  15792 S  231.5  5.1 113:46.65 java -Dap
26973 root      20   0 8920512 810820  15776 S  173.7  5.0 112:24.54 java -Dap
24386 root      20   0 8864356 702676  15764 S  98.7  4.3 295:33.69 java -Dapp
17778 root      20   0 8982272 803984  16888 S  97.4  4.9 375:15.37 java -Dapp
 1087 root      20   0 2574160 132160  31928 S  25.6  0.8   1637:21 /usr/bin/k
25589 root      20   0 8839392 585348  15772 S  20.8  3.6 160:58.44 java -Dapp
 1095 root      20   0  998512  86168  13100 S   6.5  0.5 837:37.56 /usr/bin/d
29226 root      20   0 8906120 881632  13700 S   5.8  5.4 760:36.91 java -Dapp
28206 root      20   0 7960552 341564  15700 S   4.9  2.1  66:28.23 java -Dapp
 9844 root      20   0 1632416  47092  16676 S   2.9  0.3 559:35.51 calico-nod
 9646 polkitd   20   0 4327012  97744   4752 S   2.6  0.6  25:26.93 /usr/local
```

你看，杂七杂八的应用都弄到这一台机器上了，非常不符合正常的架构。

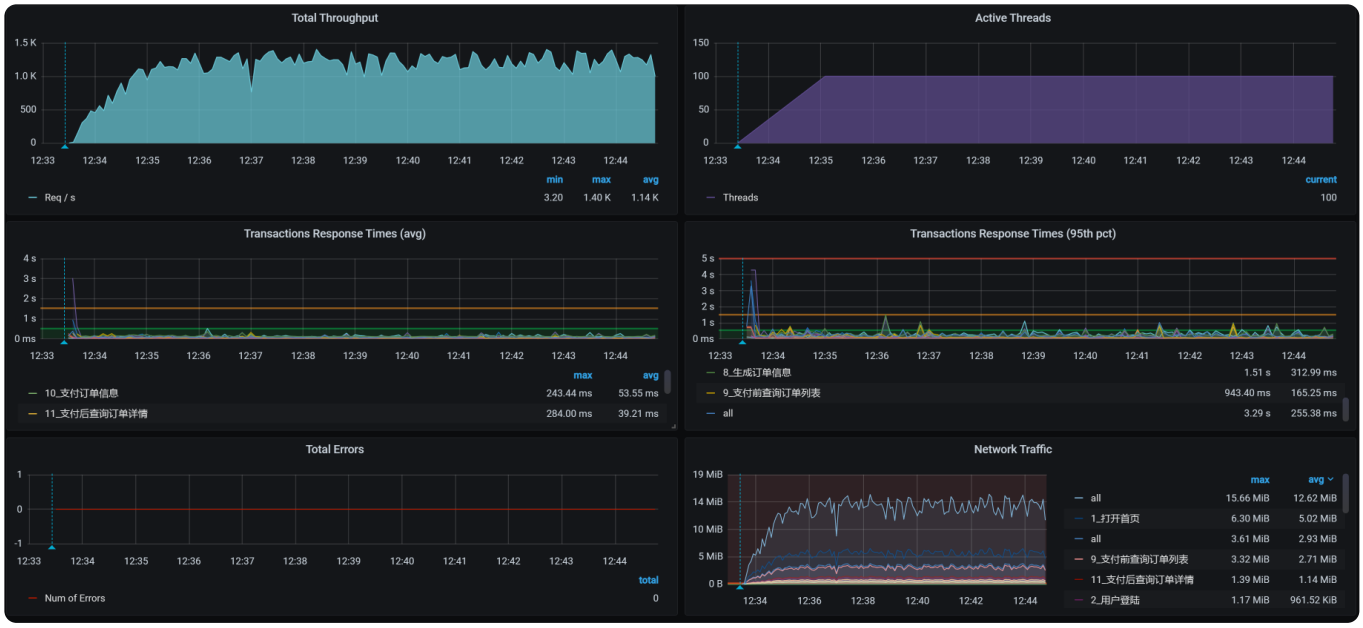
我们在前面提到，在拆分响应时间的过程中，发现是 Order 服务消耗的时间多。而 Order 服务又是当前这个场景中最需要资源的应用，那我们就先把 Auth、Portal 之类的服务移走。

移走了一些服务之后，我们先看看场景执行数据如何，再来决定是否需要再进行定向监控分析。

为了确保我们瓶颈判断的方向正确，我们先来看一下全局监控数据：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.69 week	7.64 GiB	4	1.05	12.58%	39.21%	29.95%	18.41 kB/s	113.15 kB/s	312	44	411.14 kbps	411.14 kbps
k8s-master-2	k8s-master-2	8.38 week	7.64 GiB	4	0.49	23.03%	22.86%	29.95%	0 B/s	87.54 kB/s	260	52	271.59 kbps	423.22 kbps
k8s-master-3	k8s-master-3	1.69 week	7.64 GiB	4	0.92	10.80%	35.04%	29.95%	0 B/s	92.79 kB/s	303	118	310.42 kbps	493.34 kbps
k8s-worker-1	k8s-worker-1	1.24 hour	15.51 GiB	6	5.31	52.20%	27.37%	41.70%	685.05 kB/s	5.18 MB/s	30	73	65.45 Mbps	65.71 Mbps
k8s-worker-2	k8s-worker-2	1.85 day	15.51 GiB	6	11.34	61.48%	77.81%	51.34%	27.03 kB/s	160.94 kB/s	27	1 K	23.77 Mbps	13.41 Mbps
k8s-worker-3	k8s-worker-3	1.69 week	15.51 GiB	6	11.25	60.51%	31.22%	45.17%	13.81 kB/s	161.15 kB/s	28	929	27.83 Mbps	27.02 Mbps
k8s-worker-4	k8s-worker-4	1.38 week	15.51 GiB	6	14.27	73.66%	35.85%	45.12%	18.41 kB/s	495.42 kB/s	47	29 K	42.52 Mbps	41.16 Mbps
k8s-worker-5	k8s-worker-5	7.29 week	15.51 GiB	6	3.51	47.19%	69.47%	38.11%	23.02 kB/s	9.93 kB/s	28	146	15.96 Mbps	21.06 Mbps
k8s-worker-6	k8s-worker-6	7.29 week	15.51 GiB	6	28.00	88.99%	31.08%	40.94%	23.02 kB/s	69.72 kB/s	28	377	33.78 Mbps	21.68 Mbps
k8s-worker-7	k8s-worker-7	6.99 week	15.51 GiB	4	5.48	60.69%	34.33%	42.37%	27.62 kB/s	66.31 kB/s	28	462	21.42 Mbps	16.38 Mbps
k8s-worker-8	k8s-worker-8	5.81 week	15.51 GiB	4	0.41	13.87%	33.62%	56.41%	0 B/s	3.44 kB/s	28	206	17.89 Mbps	17.99 Mbps
k8s-worker-9	k8s-worker-9	6.99 week	15.51 GiB	6	1.97	25.40%	28.30%	29.95%	9.21 kB/s	12.47 kB/s	28	77	42.00 Mbps	46.52 Mbps

再看一下场景运行数据：



你看，TPS 还挺正常的，对不对？看来我们不用再继续定向分析了。不过，这样的结果是不是真的正常了呢？我们的容量场景是不是可以结束了呢？欲知后事如何，且听下回分解。

总结

我们这节课的分析过程还是挺坎坷的，光是场景运行就执行了三次，可见即便基准容量已经测试通过了，容量场景也没那么容易运行起来。

在第一阶段的分析中，我们主要定位了在压力线程不变的情况下，TPS 随时间增加而增加的问题。请你注意，**压力不变时，TPS 平稳才是正常的**。

在第二阶段分析中，我们定位了索引不合理的问题。这样的问题算是比较常见的，定位起来比较简单，优化起来也比较容易出效果。

在第三阶段分析中，我们解决了资源使用过于集中的问题，其中，我做了资源的重新分配，并且也看起来挺正常了。

对于资源的分配，我们要作出明确的判断。在 Kubernetes 中，资源的均衡分配很多时候都需要依赖 Kubernetes 的调度能力。而 Kubernetes 的调度是需要在调度的那个时刻来判断的，在压力持续的过程中，资源消耗并不一定合理，因此，我们需要在压力持续的过程中对 Pod 做出调整。

课后作业

这就是今天的全部内容，最后给你留两个思考题吧：

1. 为什么在压力线程不变的情况下，TPS 曲线下降、响应时间上升是不合理的？
2. 当资源使用过于集中的时候，如何定位 Pod 相互之间的影响？你有没有和这节课讲的不一样的招？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果这节课让你有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 决定容量场景成败的关键因素有哪些？

下一篇 25 | 容量场景之二：缓存对性能会有什么样的影响？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。